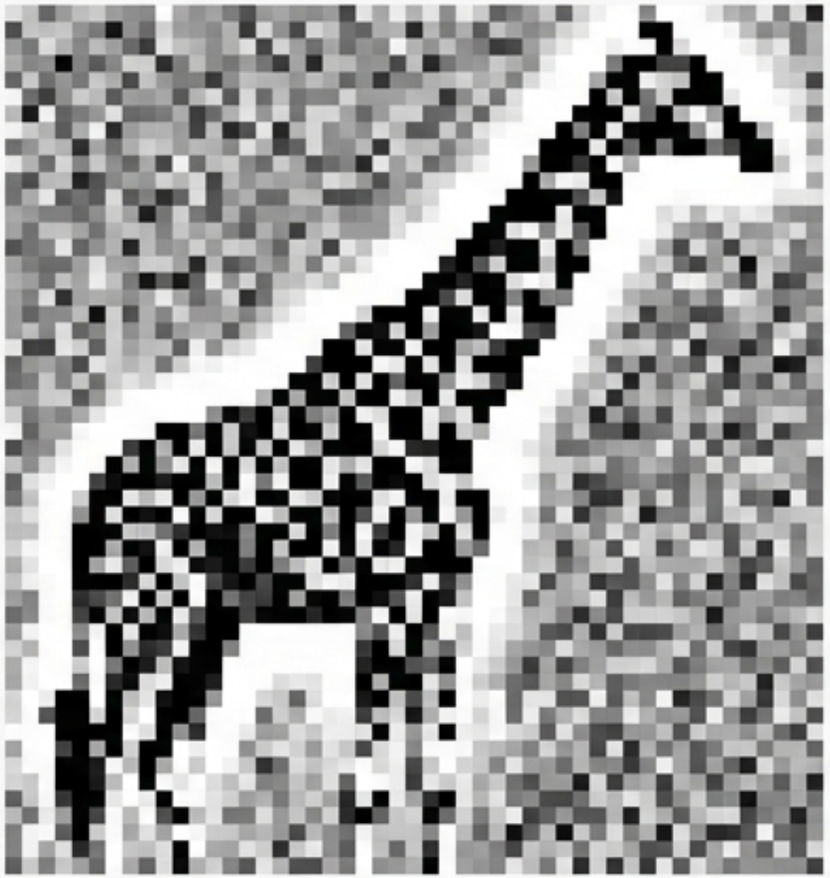
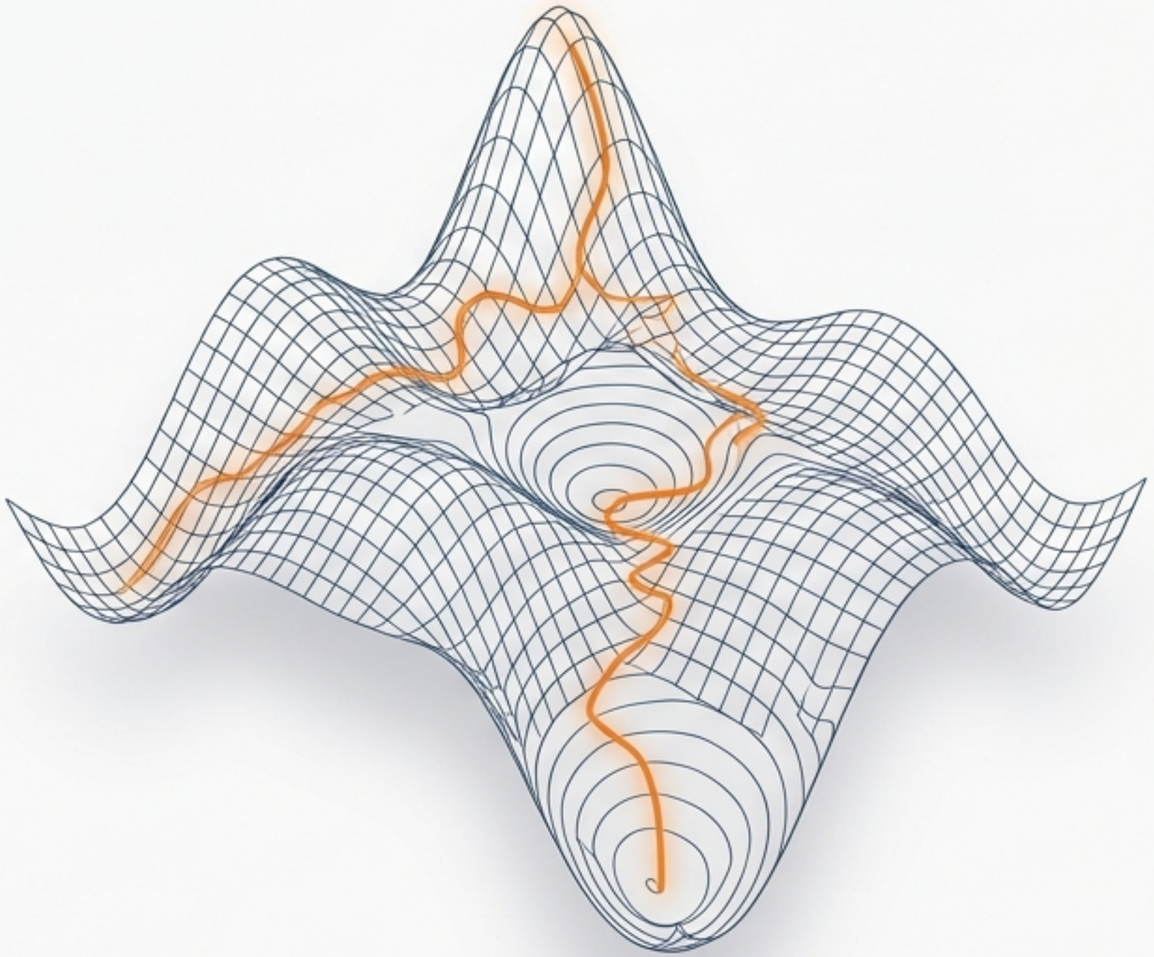
Classifiers & Vector Calculus

The Geometry of Machine Learning and the Mathematics of Optimization





Lecture Notes: Chapter 3

Chapter 3: Classifiers and Vector Calculus

Comprehensive Lecture Notes

Course: Machine Learning - Math and Architectures of Deep Learning
Textbook: Krishnendu Chaudhury
Topics: Geometric View of Classification, Gradient Descent, Loss Functions, PyTorch Implementation

---

Table of Contents

---

1. Geometric View of Image Classification

1.1 Input Representation

Images can be represented as high-dimensional vectors through a process called **rasterization**.

Example: A 224 × 224 grayscale image:

- Matrix representation: 224 rows × 224 columns
- Each element $X_{i,j} \in [0, 255]$ represents a pixel intensity
- Total elements: 224 × 224 = 50,176 pixels

Rasterization Process:

- Iterate through matrix elements left-to-right, top-to-bottom
- Store successive elements in a vector
- Result: Vector $\vec{x} \in \mathbb{R}^{50176}$

$$\vec{x} = \begin{bmatrix} x_0 = X_{0,0} \\ x_1 = X_{0,1} \\ \vdots \\ x_{223} = X_{0,223} \\ x_{224} = X_{1,0} \\ \vdots \\ x_{50175} = X_{223,223} \end{bmatrix}$$

Key Insight: Each image becomes a **point** in a 50,176-dimensional space.

1.2 Point Clusters in Feature Space

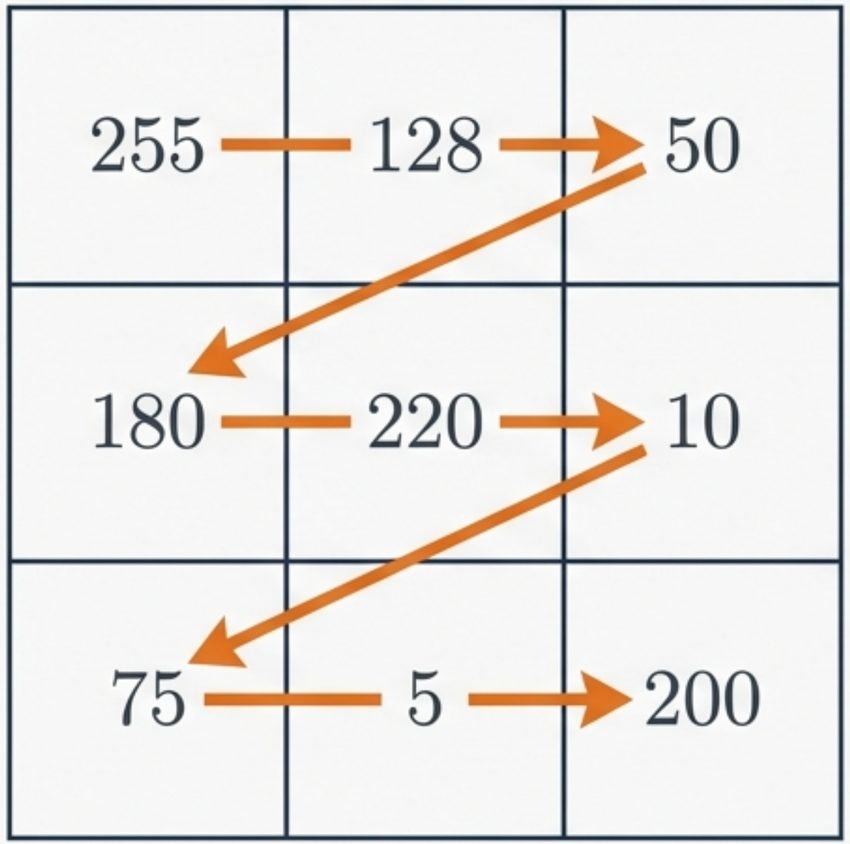
Images of the same class (e.g., "giraffe" or "car") form **clusters** in the high-dimensional feature space because:

- Members of a class share inherent commonalities
- Giraffes: predominantly yellow with black patterns
- Cars: fixed general shape
- Similar objects → similar pixel values → nearby points in space

Classification Goal: Find surfaces that separate these point clusters.

Input Representation: The Rasterization Process

Image Matrix (2D)



Feature Vector (1D)

255
128
50
180
220
10
75
5
200

The Challenge: Computers view images as matrices of brightness values, but ML models operate on vectors.

The Solution: Rasterization. We unroll the matrix row-by-row into a single flat list.

The Scale: A standard 224x224 image becomes a vector with 50,176 elements. Thus, every image is just single point coordinate in a 50,176-dimensional hyper-space.

2. Classifiers as Decision Boundaries

2.1 Geometric Definition

Definition: A classifier is a **hypersurface** in the feature space that separates point clusters corresponding to different classes.

This hypersurface is called a **decision boundary**:

- Points on one side → Class 1
- Points on other side → Class 2

2.2 Types of Decision Boundaries

Linear Classifier (Hyperplane)

For simple, well-separated classes:

$$\phi(\vec{x}; \vec{w}, b) = \vec{w}^T \vec{x} + b = 0$$

- $\vec{w}$: weight vector (normal to the hyperplane)
- b: bias term
- Decision rule: $\text{sign}(\phi(\vec{x}; \vec{w}, b))$ determines the class

Example: Car vs. Giraffe classification (Figure 3.1a from textbook)

- Points marked 'g' for giraffe cluster on one side
- Points marked 'c' for car cluster on other side
- Linear decision boundary separates them

Nonlinear Classifier (Hypersphere)

For more complex separations:

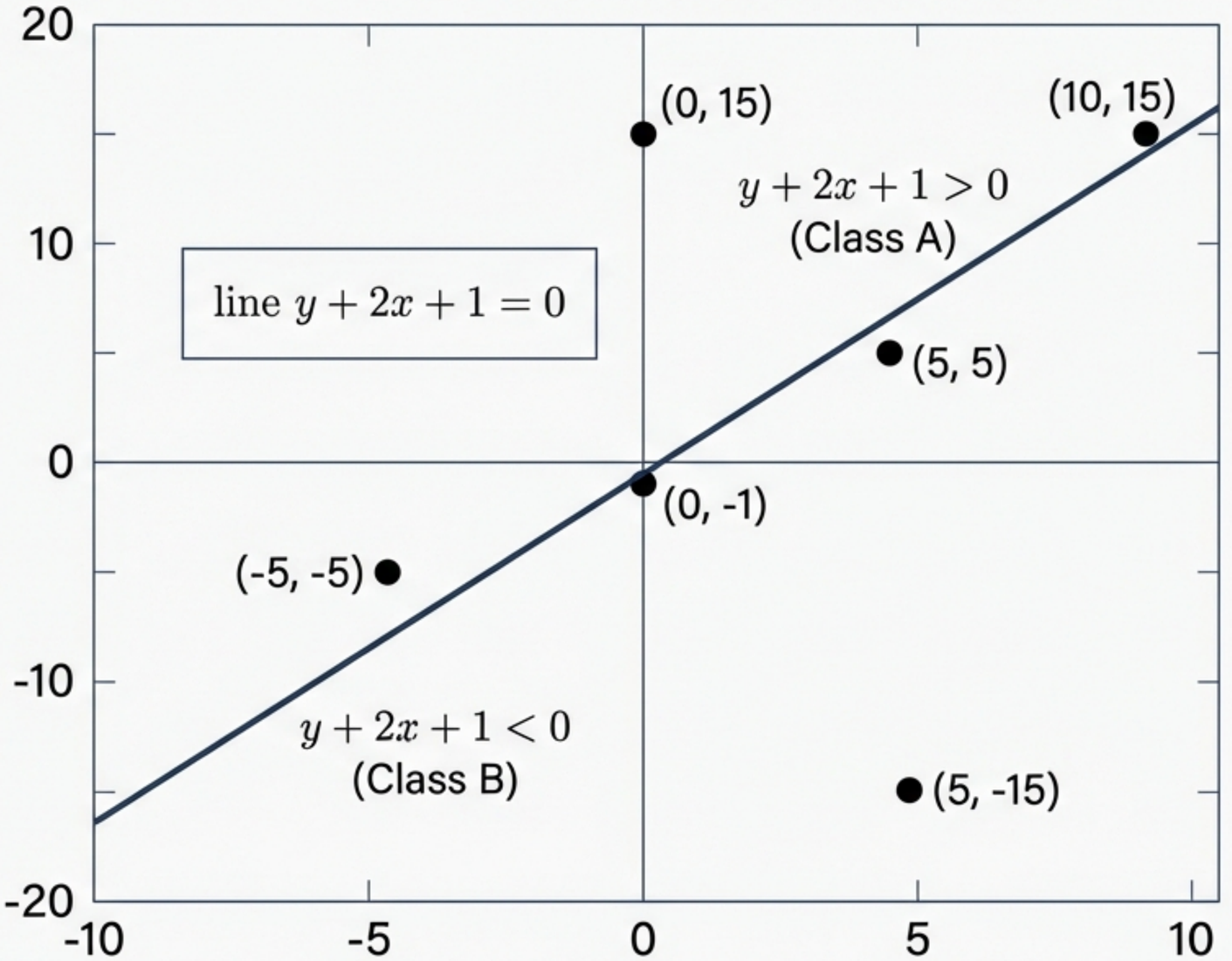
$$\phi(\vec{x}; \vec{w}, b) = \vec{x}^T \begin{bmatrix} w_0 & 0 & \cdots & 0 \\ 0 & w_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w_n \end{bmatrix} \vec{x} + b = 0$$

Classifiers are Decision Boundaries

Clustering: Objects of the same class share pixel patterns, grouping together in vector space.

The Classifier:
Geometrically, a classifier is a hypersurface (*Hyperplane*) that slices the space to separate these clusters.

The Logic:
Side A → Class: *Car*
Side B → Class: *Giraffe*



This represents a **sphere** (or hypersphere in higher dimensions).

Example: Horse vs. Zebra classification (Figure 3.1b from textbook)

- Points marked 'h' for horse cluster
- Points marked 'z' for zebra cluster
- Circular decision boundary needed

2.3 Sign of the Surface Function

For **binary classifiers**, the sign of $\phi(\vec{x}; \vec{w}, b)$ has special significance:

Example: Line in 2D: $y + 2x + 1 = 0$

- Points where $y + 2x + 1 = 0$: **on the line**
- Points where $y + 2x + 1 > 0$: **one half-plane**
- Points where $y + 2x + 1 < 0$: **other half-plane**

Decision Rule:

```
if φ(x⃗; w⃗, b) > 0:
    return Class 1
else:
    return Class 2
```

---

3. Modeling in a Nutshell

3.1 The Problem Setup

Given:

- Training inputs: $\{\vec{x}_i\}_{i=1}^n$
- Ground truth labels: $\{\bar{y}_i\}_{i=1}^n$
- Training dataset: $\{(\vec{x}_i, \bar{y}_i)\}_{i=1}^n$

Goal: Find the decision boundary that best separates the training data.

The Mathematical Model

Weights (Orientation)
Determines the angle of the boundary.

Bias (Position)
Determines the offset from the origin.

$$q(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}^T \mathbf{x} + b$$

Input Vector

Binary Classification Logic:

If **result** > 0 : Predict Class 1

If **result** < 0 : Predict Class 2

Note: We choose the linear family (architecture), but we must search for the specific $\mathbf{w}$ and b values (training).

3.2 Two-Step Modeling Process

Step 1: Model Architecture Selection

Choose a parametric function family $\phi(\vec{x}; \vec{w}, b)$:

For simple problems: Linear model

$$\phi(\vec{x}; \vec{w}, b) = \vec{w}^T \vec{x} + b$$

For complex problems: Nonlinear models (polynomials, neural networks, etc.)

Note: Parameters $\vec{w}, b$ are initially **unknown**.

Step 2: Model Training

Objective: Estimate parameters $\vec{w}, b$ to minimize training error.

Process:

1. For each training instance $\vec{x}_i$, compute: $y_i = \phi(\vec{x}_i; \vec{w}, b)$
2. Calculate error: $e_i = \|y_i - \bar{y}_i\|$
3. Aggregate training error: $L(\vec{w}, b) = \sum_{i=1}^{n} e_i^2$
4. **Iteratively adjust** $\vec{w}, b$ to reduce $L(\vec{w}, b)$

3.3 Creating Ground Truth Labels

Manual Labeling: Often the most time-consuming part of machine learning

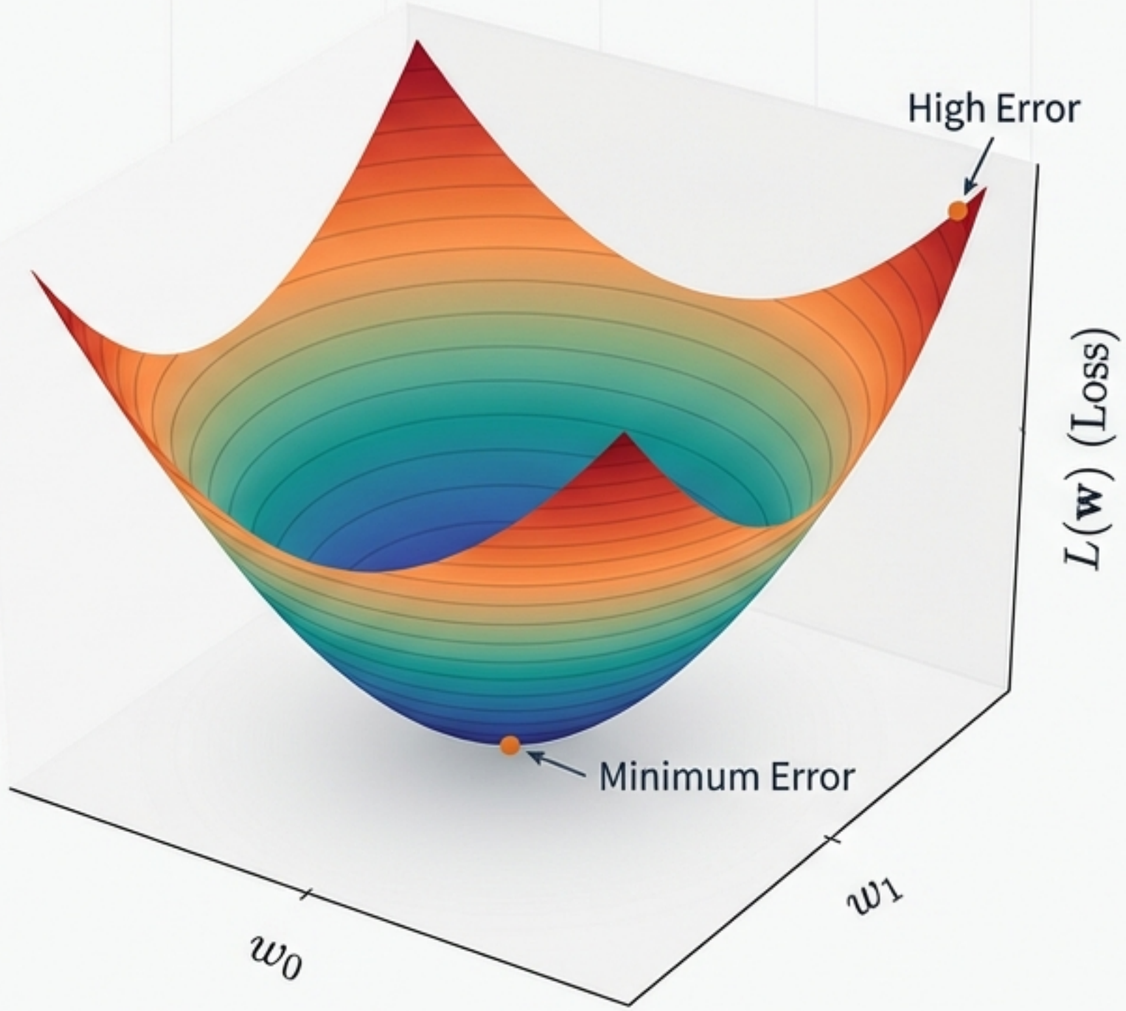
- Human annotators label each training image
- Example: Look at image → label as "car" or "giraffe"
- This is why active research focuses on reducing labeling requirements

---

4. Error/Loss Functions

Quantifying Failure: The Loss Function

- **The Goal:** Match model output y to ground truth $\hat{y}$.
- **Squared Error:** $e^{(i)} = \left(q(\mathbf{x}^{(i)}) - \hat{y}^{(i)}\right)^2$
- **Total Loss:** $L(\mathbf{w}, b) = \sum \left(e^{(i)}\right)^2$
- **The Landscape:** Plotting Loss against Weight values creates an error surface. Training is the process of descending this surface.



4.1 Definition

The **loss function** (or error function) quantifies how far the model's predictions are from the known correct outputs.

For a single training instance $(\vec{x}_i, \bar{y}_i)$:

$$L_i = \left(\phi(\vec{x}_i; \vec{w}, b) - \bar{y}_i\right)^2$$

Aggregate loss over all training data:

$$L(\vec{w}, b) = \sum_{i=0}^{N} \left(\phi(\vec{x}_i; \vec{w}, b) - \bar{y}_i\right)^2$$

4.2 Detailed Workout Example

Problem: Fit a linear model $y = wx + b$ to data points.

Given Data: | x_i | $\bar{y}_i$ (true) | |-------|------------------| | 1.0 | 3.0 | | 2.0 | 5.0 | | 3.0 | 7.0 | | 4.0 | 9.0 | | 5.0 | 11.0 |

Model: $y = wx + b$

Initial guess: $w = 1.5, b = 1.0$

Step-by-step Calculation:

Predictions:

- $\hat{y}_1 = 1.5(1.0) + 1.0 = 2.5$
- $\hat{y}_2 = 1.5(2.0) + 1.0 = 4.0$
- $\hat{y}_3 = 1.5(3.0) + 1.0 = 5.5$
- $\hat{y}_4 = 1.5(4.0) + 1.0 = 7.0$
- $\hat{y}_5 = 1.5(5.0) + 1.0 = 8.5$

Squared Errors:

- $e_1^2 = (2.5 - 3.0)^2 = 0.25$
- $e_2^2 = (4.0 - 5.0)^2 = 1.00$

- $e_3^2 = (5.5 - 7.0)^2 = 2.25$
- $e_4^2 = (7.0 - 9.0)^2 = 4.00$
- $e_5^2 = (8.5 - 11.0)^2 = 6.25$

Total Loss:

$$L(w = 1.5, b = 1.0) = 0.25 + 1.00 + 2.25 + 4.00 + 6.25 = 13.75$$

Optimal Solution (analytically): $w = 2.0, b = 1.0$

Verification:

- $\hat{y}_1 = 2.0(1.0) + 1.0 = 3.0$ ✓
- $\hat{y}_2 = 2.0(2.0) + 1.0 = 5.0$ ✓
- $\hat{y}_3 = 2.0(3.0) + 1.0 = 7.0$ ✓
- $\hat{y}_4 = 2.0(4.0) + 1.0 = 9.0$ ✓
- $\hat{y}_5 = 2.0(5.0) + 1.0 = 11.0$ ✓

Total Loss at Optimal: $L(w = 2.0, b = 1.0) = 0$ ✓

---

5. Gradient Vectors

5.1 Core Training Algorithm

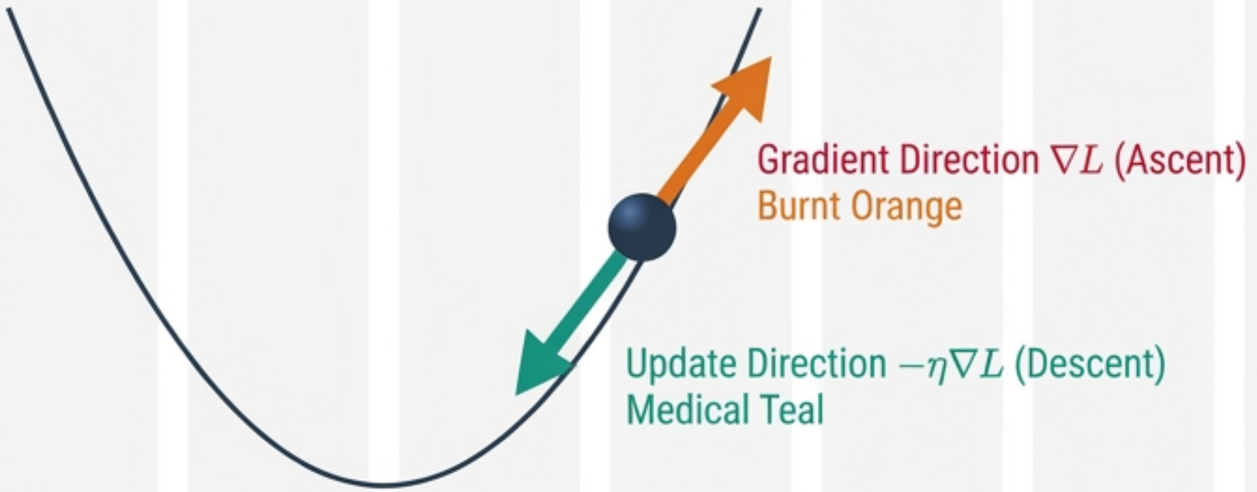
Goal: Minimize loss $L(\vec{w}, b)$ by adjusting parameters iteratively.

Key Idea: The **gradient** indicates the direction of maximum increase in a function.

$$\nabla_{\vec{w},b} L(\vec{w}, b) = \text{direction of steepest ascent}$$

To minimize: Move in the **opposite direction** (negative gradient)

The Strategy: Gradient Descent



Iterative Optimization:

1. We are "blind hikers" in a fog. We cannot see the bottom, only the slope under our feet.
2. The **Gradient** points uphill (steepest ascent).
3. The **Update Rule**: Step in the *opposite* direction.

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \nabla L$$

(η = Learning Rate/Step Size)

5.2 Training Algorithm

```
Algorithm: Gradient Descent Training

Initialize w⃗, b with random values

while L(w⃗, b) > threshold:
    # Compute gradient
    g⃗ = ∇_w⃗,b L(w⃗, b)

    # Update parameters
    w⃗ = w⃗ - η·g⃗_w
    b = b - η·g⃗_b

    # Recompute loss
    L = compute_loss(w⃗, b)

return w⃗*, b*
```

Parameters:

- η: **learning rate** (step size)
- Large η → faster progress, risk of overshooting
- Small η → slower but more stable

Stopping Criteria:

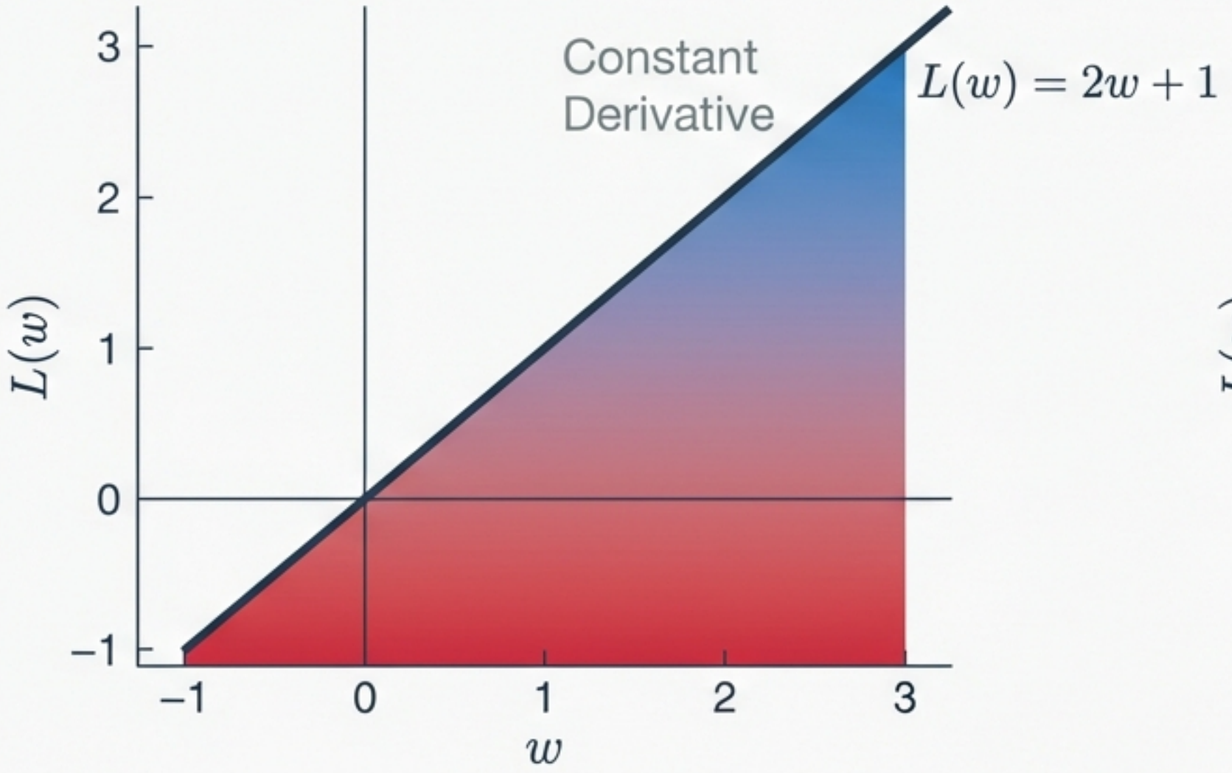
- Loss below threshold
- Gradient near zero
- Maximum iterations reached

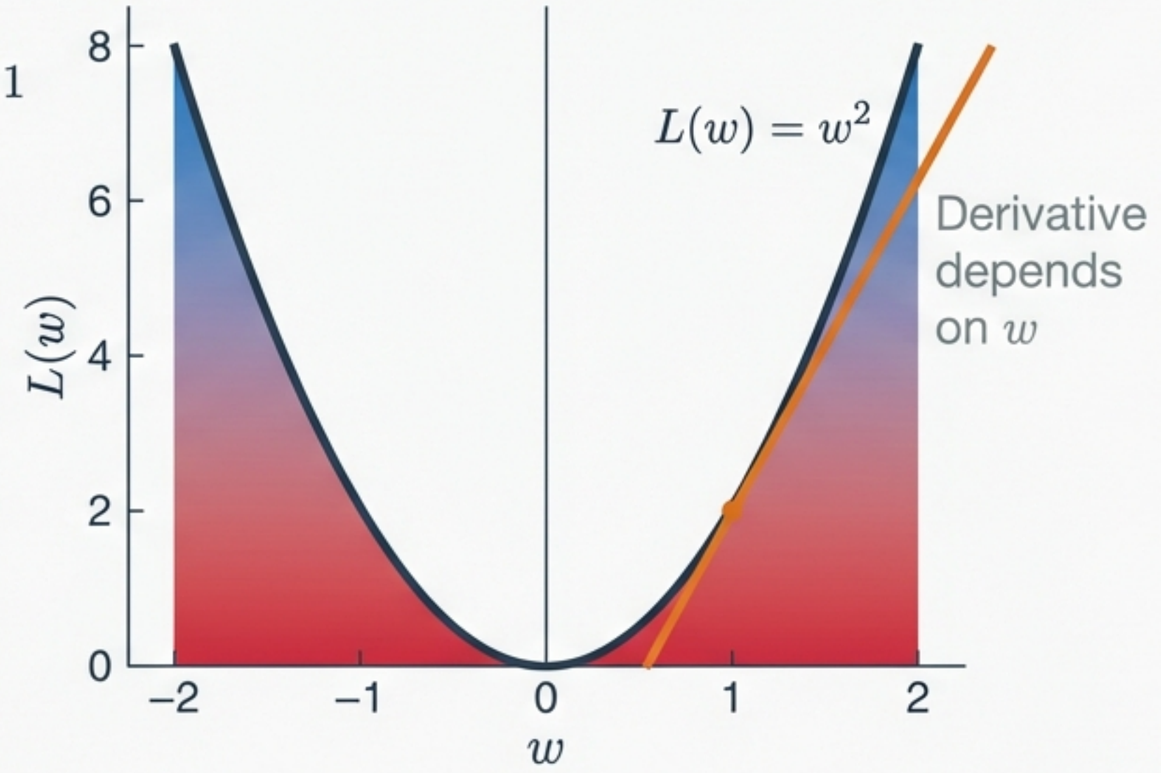
5.3 Why Gradients Work

Geometric Intuition:

- Loss function defines a surface in parameter space
- Gradient at any point is **perpendicular** to level curves
- Points in direction of **maximum change**
- Following negative gradient → steepest descent

Gradient Mechanics: 1D Intuition





- **Case A (Slope > 0):** Increasing w increases Loss. To reduce Loss, move Left (decrease w).
- **Case B (Slope < 0):** Increasing w decreases Loss. To reduce Loss, move Right (increase w).
- **Universal Rule:** The update $\Delta w = -\eta \frac{dL}{dw}$ automatically satisfies both cases.

Mathematical Justification: Taylor series approximation (covered in Section 11)

---

6. Gradients: Machine Learning-Centric Introduction

6.1 The Fundamental Question

Given loss function $L(\vec{w})$ and current parameters $\vec{w}$:

Question: What change $\Delta\vec{w}$ will **maximally reduce** the loss?

Equivalently: What $\Delta\vec{w}$ makes $\Delta L = L(\vec{w} + \Delta\vec{w}) - L(\vec{w})$ most negative?

Answer: The gradient $\nabla_{\vec{w}} L(\vec{w})$ provides this information.

6.2 Gradient Properties

Definition: Vector of all partial derivatives

$$\nabla L(\vec{w}) = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

Properties:

1. **Direction**: Points toward maximum increase
2. **Magnitude**: Rate of maximum change
3. **Zero at extrema**: $\nabla L(\vec{w}^*) = \vec{0}$ at minimum/maximum

---

Scaling Up: The Gradient Vector

Partial Derivatives:

How does Loss change if I adjust *only* w_i?

$$\left(\frac{\partial L}{\partial w_i}\right)$$

The Gradient Vector:

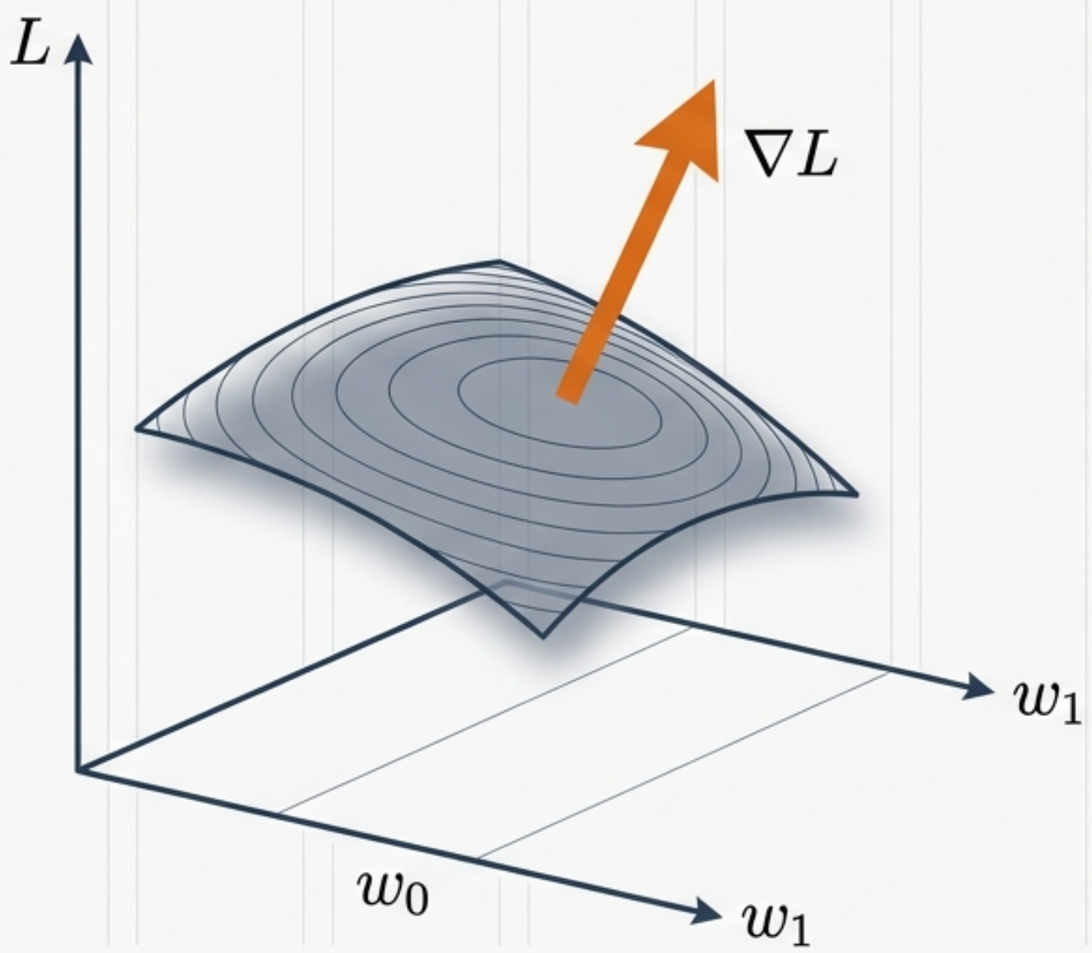
Collects all partials.

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \vdots \end{bmatrix}.$$

Dot Product Rule:

Total change $\Delta L \approx \nabla L \cdot \Delta \mathbf{w}$.

To get the maximum decrease in Loss, the step $\Delta \mathbf{w}$ must be anti-parallel to the gradient vector.



7. One-Dimensional Loss Functions

7.1 Linear Loss Function

Example: $L(w) = mw + c$

Change in loss:

$$\Delta L = L(w + \Delta w) - L(w) = m\Delta w$$

Rate of change:

$$\frac{\Delta L}{\Delta w} = m$$

Derivative:

$$\frac{dL}{dw} = m \text{ (constant everywhere)}$$

Update rule:

$$\Delta L = \frac{dL}{dw}\Delta w$$

7.2 Parabolic Loss Function (Workout Example)

Example: $L(w) = w^2$

Given: Current position $w = 3$

Step 1: Compute derivative

$$\frac{dL}{dw} = 2w$$

At $w = 3$:

$$\left.\frac{dL}{dw}\right|_{w=3} = 2(3) = 6$$

Step 2: Choose learning rate $\eta = 0.1$

Step 3: Update parameter

$$w_{\text{new}} = w - \eta \frac{dL}{dw} = 3 - 0.1(6) = 2.4$$

Step 4: Verify loss decreased

- $L(3) = 9$
- $L(2.4) = 5.76$
- $\Delta L = 5.76 - 9 = -3.24$ ✓ (decreased)

Repeat until $w \approx 0$ (minimum)

7.3 General 1D Update Rule

For any smooth function $L(w)$:

$$\Delta L = \frac{dL}{dw} \Delta w$$

To minimize:

- If $\frac{dL}{dw} > 0$: move left ($\Delta w < 0$)
- If $\frac{dL}{dw} < 0$: move right ($\Delta w > 0$)

Geometric interpretation: Follow the tangent downward

---

8. Multidimensional Loss Functions

8.1 The Challenge

With multiple parameters $\vec{w} = [w_0, w_1, \ldots, w_n]^T$:

Key difference from 1D:

- Parameter change $\Delta \vec{w}$ is a **vector**

- Has both magnitude $||\Delta\vec{w}||$ and direction $\hat{\Delta w}$
- Same magnitude, different direction → different change in L

8.2 Example: 2D Loss Function

Function: $L(w_0, w_1) = 2w_0^2 + 3w_1^2$

Current point: $\vec{w} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$

Loss at point: $L(3, 4) = 2(9) + 3(16) = 66$

Displacement 1: $\Delta\vec{w} = \begin{bmatrix} 0.0003 \\ 0.0004 \end{bmatrix}$

New point: $\vec{w}' = \begin{bmatrix} 3.0003 \\ 4.0004 \end{bmatrix}$

New loss:

$$L(3.0003, 4.0004) = 2(3.0003)^2 + 3(4.0004)^2 \approx 66.0132$$

Change: $\Delta L_1 = 0.0132$

Displacement 2: $\Delta\vec{w} = \begin{bmatrix} 0.0004 \\ 0.0003 \end{bmatrix}$

New point: $\vec{w}' = \begin{bmatrix} 3.0004 \\ 4.0003 \end{bmatrix}$

New loss:

$$L(3.0004, 4.0003) = 2(3.0004)^2 + 3(4.0003)^2 \approx 66.0120$$

Change: $\Delta L_2 = 0.0120$

Observation: Same magnitude $||\Delta\vec{w}|| = 0.0005$, but different directions → different changes in loss!

---

9. Partial Derivatives and Gradients

9.1 Partial Derivatives

Definition: Derivative with respect to one variable, treating others as constants.

Example: $L(w_0, w_1) = 2w_0^2 + 3w_1^2$

$$\frac{\partial L}{\partial w_0} = 4w_0$$

$$\frac{\partial L}{\partial w_1} = 6w_1$$

9.2 Total Change via Partial Derivatives

For displacement $\Delta\vec{w} = \begin{bmatrix} \Delta w_0 \\ \Delta w_1 \\ \vdots \\ \Delta w_n \end{bmatrix}$:

$$\Delta L = \frac{\partial L}{\partial w_0}\Delta w_0 + \frac{\partial L}{\partial w_1}\Delta w_1 + \cdots + \frac{\partial L}{\partial w_n}\Delta w_n$$

Compact form:

$$\Delta L = \nabla L(\vec{w})^T \Delta\vec{w} = \nabla L(\vec{w}) \cdot \Delta\vec{w}$$

9.3 Gradient Definition

$$\nabla L(\vec{w}) = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

9.4 Workout Example

Function: $L(w_0, w_1) = 2w_0^2 + 3w_1^2$

Point: $(w_0, w_1) = (3, 4)$

Step 1: Compute partial derivatives

$$\frac{\partial L}{\partial w_0} = 4w_0 = 4(3) = 12$$
$$\frac{\partial L}{\partial w_1} = 6w_1 = 6(4) = 24$$

Step 2: Form gradient

$$\nabla L(3, 4) = \begin{bmatrix} 12 \\ 24 \end{bmatrix}$$

Step 3: Choose learning rate $\eta = 0.1$

Step 4: Update parameters

$$\begin{bmatrix} w_0 \\ w_1 \end{bmatrix}_{\text{new}} = \begin{bmatrix} 3 \\ 4 \end{bmatrix} - 0.1 \begin{bmatrix} 12 \\ 24 \end{bmatrix} = \begin{bmatrix} 1.8 \\ 1.6 \end{bmatrix}$$

Step 5: Verify improvement

- Old loss: $L(3, 4) = 2(9) + 3(16) = 66$
- New loss: $L(1.8, 1.6) = 2(3.24) + 3(2.56) = 14.16$
- Improvement: $66 - 14.16 = 51.84$ ✓

9.5 The Gradient is Zero at Minimum

Example: $L(w_0, w_1) = \sqrt{w_0^2 + w_1^2}$

Gradient:

$$\nabla L = \frac{1}{2\sqrt{w_0^2 + w_1^2}} \begin{bmatrix} 2w_0 \\ 2w_1 \end{bmatrix} = \frac{1}{\sqrt{w_0^2 + w_1^2}} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

Set equal to zero:

$$\nabla L = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Solution: $w_0 = 0, w_1 = 0$

Verification: Minimum is at origin with $L(0, 0) = 0$ ✓

---

10. Level Surface Representation and Loss Minimization

10.1 Level Contours and Surfaces

Definition: Set of points where function has constant value.

2D Example: $L(w_0, w_1) = \sqrt{w_0^2 + w_1^2}$

Level contours: $w_0^2 + w_1^2 = c^2$ (concentric circles)

- $c = 1$: Circle of radius 1 centered at origin
- $c = 2$: Circle of radius 2 centered at origin
- $c = 5$: Circle of radius 5 centered at origin

3D Example: $L(w_0, w_1, w_2) = \sqrt{w_0^2 + w_1^2 + w_2^2}$

Level surfaces: $w_0^2 + w_1^2 + w_2^2 = c^2$ (concentric spheres)

10.2 Relationship Between Gradients and Level Surfaces

Key Property: At any point, the gradient is **perpendicular** to the level surface through that point.

Why this matters for optimization:

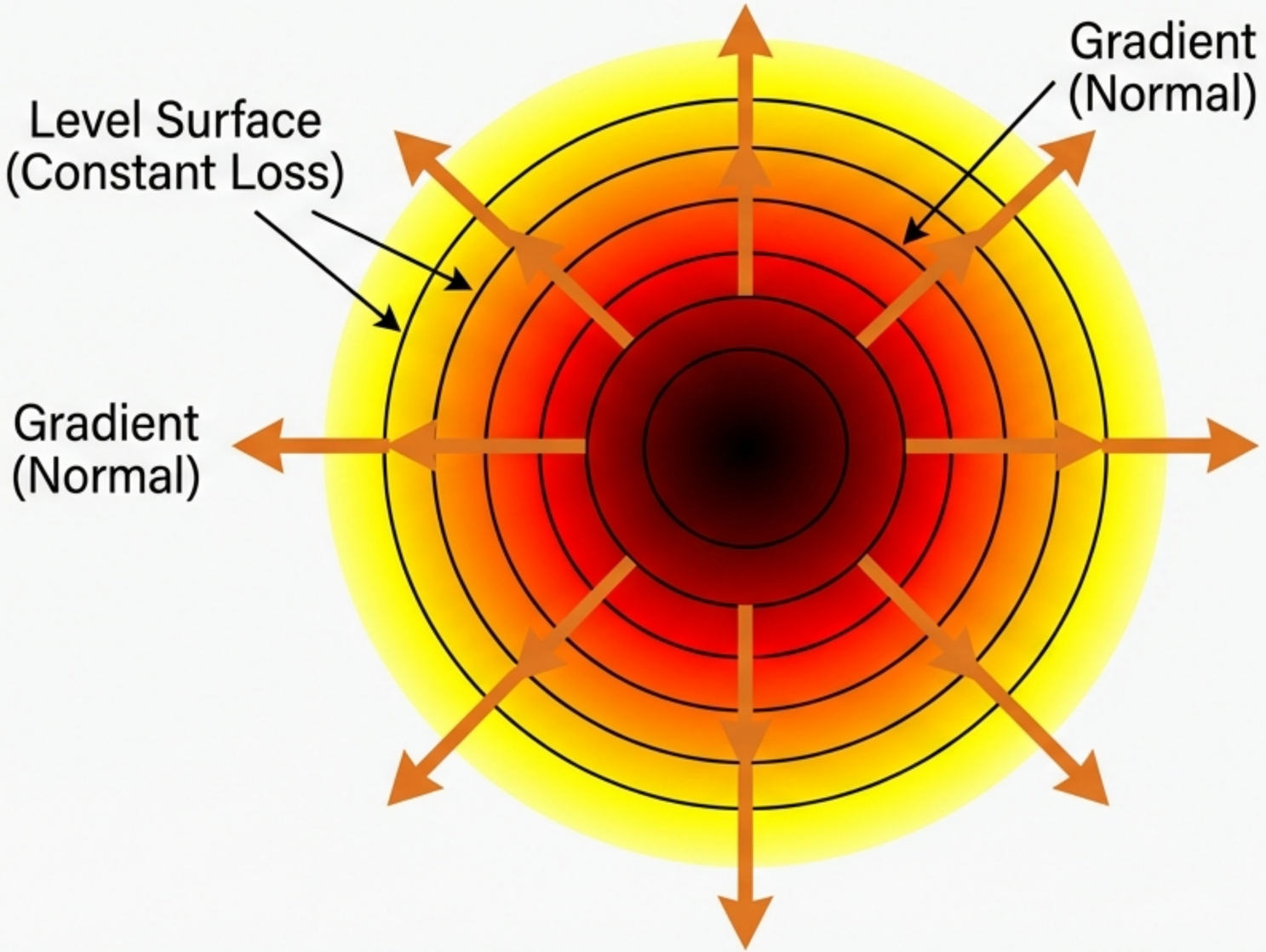
1. Moving along level surface → no change in function value
2. Moving perpendicular (along gradient) → maximum change
3. To minimize: move perpendicular to level surfaces, toward center

10.3 Detailed Explanation

Consider: $L(w_0, w_1) = w_0^2 + w_1^2$

Level contours: Circles $w_0^2 + w_1^2 = r^2$

Level Surfaces & Orthogonality



- **Level Contours:** Curves where Loss is constant. Moving along a contour = zero learning.
- **Orthogonality:** The Gradient is always perpendicular (normal) to the level surface.
- **Steepest Path:** Gradient descent cuts across these contour lines at 90-degree angles to reach the center efficiently.

Gradient at point (w_0, w_1):

$$\nabla L = \begin{bmatrix} 2w_0 \\ 2w_1 \end{bmatrix} = 2 \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

Observation: Gradient points **radially outward** from origin.

For any point on circle: Gradient is perpendicular to the circle (tangent).

Geometric interpretation:

- Tangent to circle = direction of zero change
- Normal to circle (gradient direction) = direction of maximum change
- Following gradient inward = moving toward minimum

10.4 Visualization Insights

Heat map representation:

- Color intensity represents function value
- Dark (cold) = low values → near minimum
- Bright (hot) = high values → far from minimum
- Gradient arrows point from cold to hot
- To minimize: follow arrows backward (toward cold)

Path to minimum:

- Start at arbitrary point
- Compute gradient
- Move opposite to gradient
- Repeat until gradient ≈ 0

---

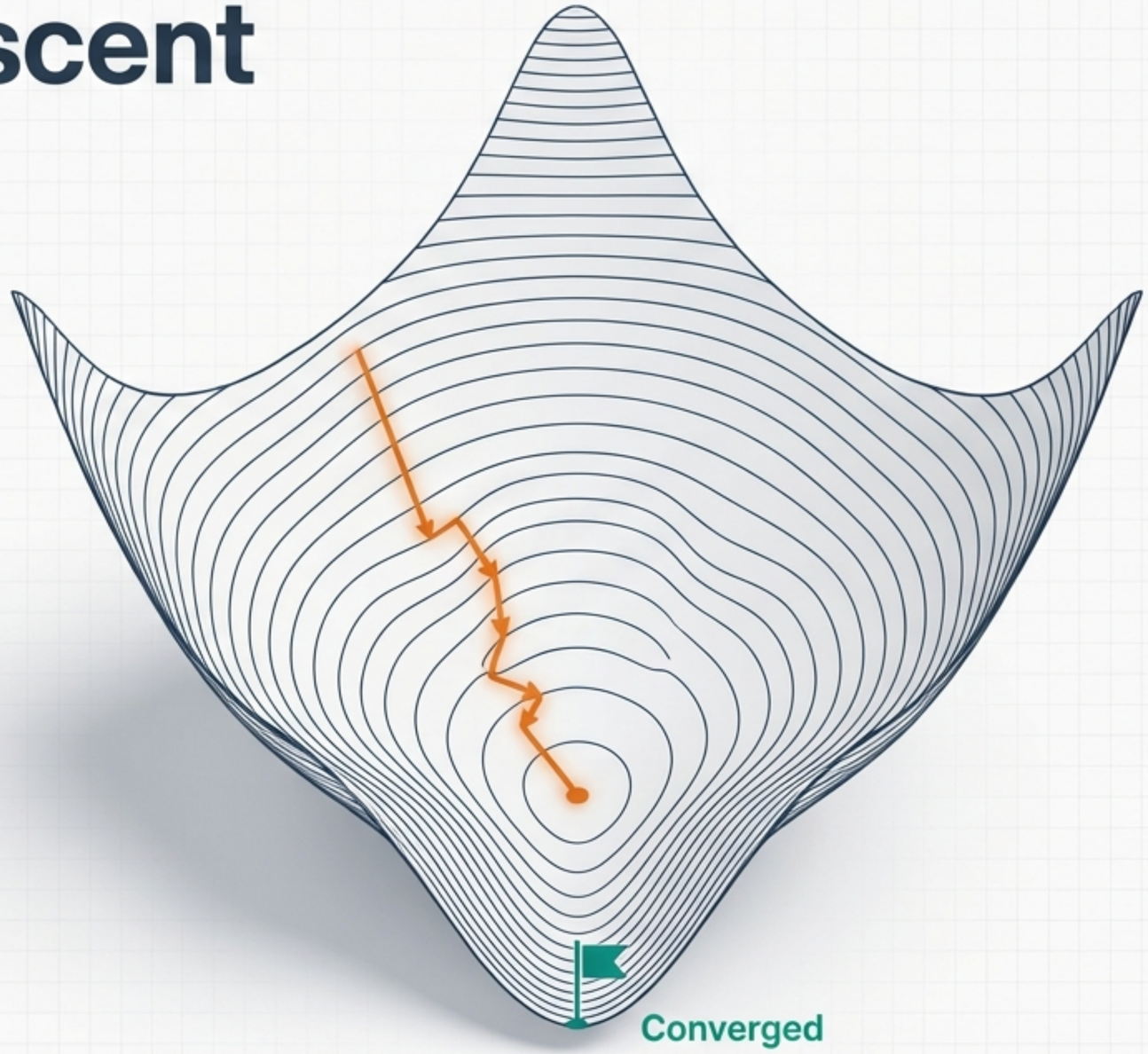
11. Local Approximation to Loss Function

The Power of Descent

We started with a pixelated image and a random guess.

By treating learning as a geometry problem—descending a valley of error—we enable machines to ‘learn’ complex patterns without explicit programming rules.

The **Gradient** is the compass that guides this journey from randomness to intelligence.



11.1 Taylor Series: One-Dimensional Case

Purpose: Approximate function near a point using derivatives.

General form:

$$L(w+\Delta w) = L(w) + \frac{\Delta w}{1!}\frac{dL}{dw} + \frac{(\Delta w)^2}{2!}\frac{d^2L}{dw^2} + \frac{(\Delta w)^3}{3!}\frac{d^3L}{dw^3} + \cdots$$

First-order approximation (linear):

$$L(w+\Delta w) \approx L(w) + \frac{dL}{dw}\Delta w$$

This is the tangent line approximation.

Second-order approximation (quadratic):

$$L(w+\Delta w) \approx L(w) + \frac{dL}{dw}\Delta w + \frac{1}{2}\frac{d^2L}{dw^2}(\Delta w)^2$$

11.2 Example: Exponential Function

Function: $f(x) = e^x$

Taylor series around $x = 0$:

Note: $\frac{d^n}{dx^n}(e^x) = e^x$ and $e^0 = 1$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Truncated approximations:

- 0th order: $e^x \approx 1$
- 1st order: $e^x \approx 1 + x$
- 2nd order: $e^x \approx 1 + x + \frac{x^2}{2}$
- 3rd order: $e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$

Accuracy improves with more terms and closer to expansion point.

Local Approximation: The Taylor Series

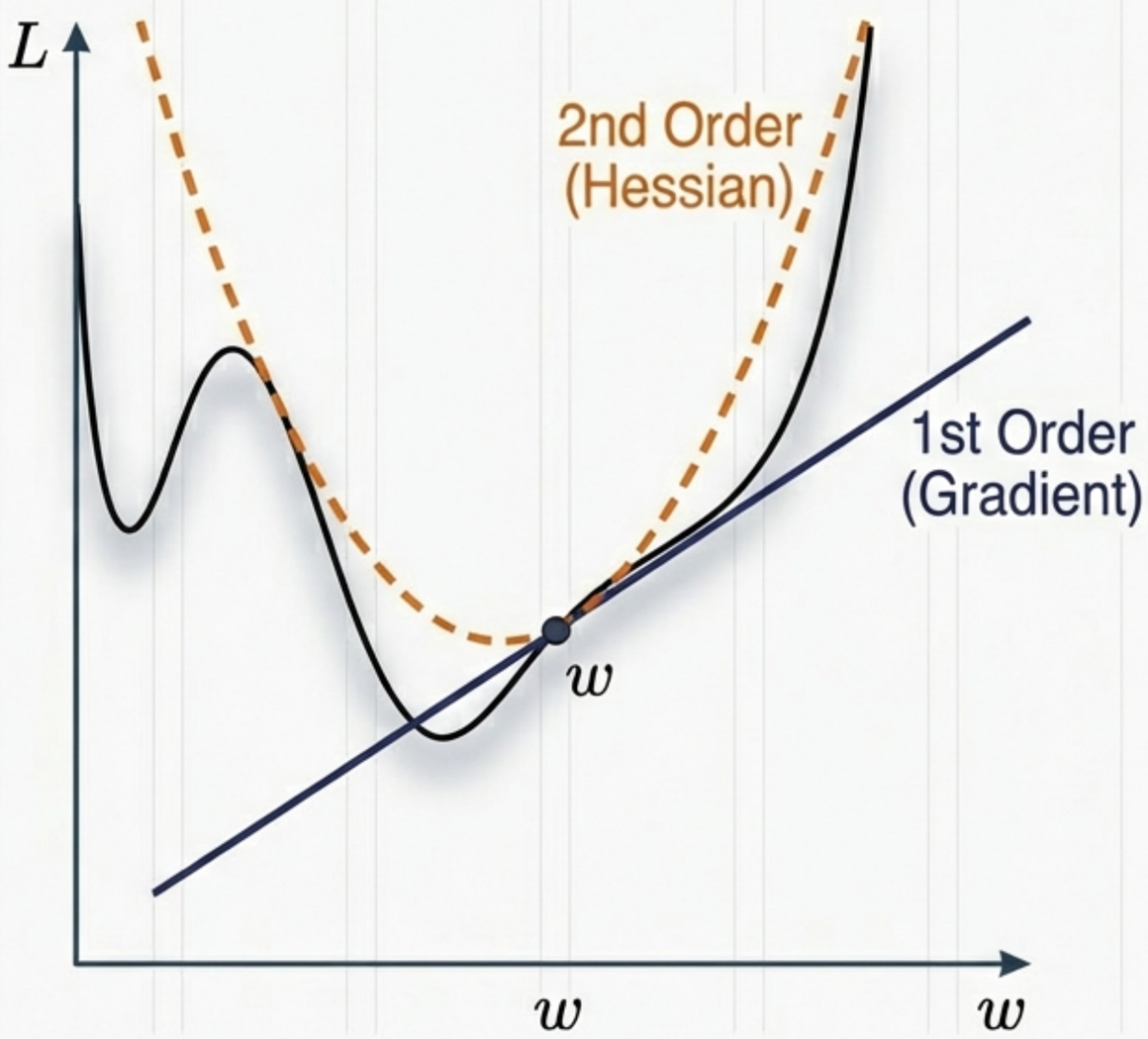
The Concept: We approximate the complex landscape locally.

First-Order Approximation (Gradient Descent):

$$L(w + \Delta w) \approx L(w) + \nabla L^T \Delta w$$

Approximates surface as a flat plane.
Valid for small steps.

Second-Order Approximation (Newton's Method):
Includes the **Hessian Matrix** (H) to account for curvature. More accurate, but computationally expensive (N^2 derivatives).



11.3 Multidimensional Taylor Series

First-order approximation:

$$L(\vec{w} + \Delta\vec{w}) \approx L(\vec{w}) + \nabla L(\vec{w})^T \Delta\vec{w}$$

This is the tangent plane approximation.

Second-order approximation:

$$L(\vec{w} + \Delta\vec{w}) \approx L(\vec{w}) + \nabla L(\vec{w})^T \Delta\vec{w} + \frac{1}{2}\Delta\vec{w}^T H[L(\vec{w})]\Delta\vec{w}$$

where H is the **Hessian matrix**.

11.4 The Hessian Matrix

Definition: Matrix of second-order partial derivatives

$$H[L(\vec{w})] = \begin{bmatrix} \frac{\partial^2 L}{\partial w_0^2} & \frac{\partial^2 L}{\partial w_0 \partial w_1} & \cdots & \frac{\partial^2 L}{\partial w_0 \partial w_n} \\ \frac{\partial^2 L}{\partial w_1 \partial w_0} & \frac{\partial^2 L}{\partial w_1^2} & \cdots & \frac{\partial^2 L}{\partial w_1 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial w_n \partial w_0} & \frac{\partial^2 L}{\partial w_n \partial w_1} & \cdots & \frac{\partial^2 L}{\partial w_n^2} \end{bmatrix}$$

Properties:

- Symmetric: $\frac{\partial^2 L}{\partial w_i \partial w_j} = \frac{\partial^2 L}{\partial w_j \partial w_i}$
- Captures curvature of loss surface
- Used in advanced optimization methods (Newton's method)

11.5 Example: 2D Quadratic Function

Function: $L(w_0, w_1) = 2w_0^2 + 3w_1^2$

First derivatives:

$$\frac{\partial L}{\partial w_0} = 4w_0, \quad \frac{\partial L}{\partial w_1} = 6w_1$$

Second derivatives:

$$\frac{\partial^2 L}{\partial w_0^2} = 4, \quad \frac{\partial^2 L}{\partial w_1^2} = 6$$

$$\frac{\partial^2 L}{\partial w_0 \partial w_1} = 0, \quad \frac{\partial^2 L}{\partial w_1 \partial w_0} = 0$$

Hessian matrix:

$$H = \begin{bmatrix} 4 & 0 \\ 0 & 6 \end{bmatrix}$$

Interpretation:

- Diagonal elements: curvature along each axis
- Off-diagonal elements: coupling between variables
- For this function: no coupling (off-diagonal = 0)

11.6 Practical Use of Taylor Series

Why it matters:

1. **First-order**: Tells us direction to move (gradient descent)
2. **Second-order**: Tells us how to adjust step size (Newton's method)
3. **Higher orders**: Rarely used (diminishing returns, computational cost)

In practice:

- Gradient descent uses first-order approximation
- Some optimizers (L-BFGS) approximate second-order information
- Pure Newton's method too expensive for deep learning

Curvature and The Hessian Matrix

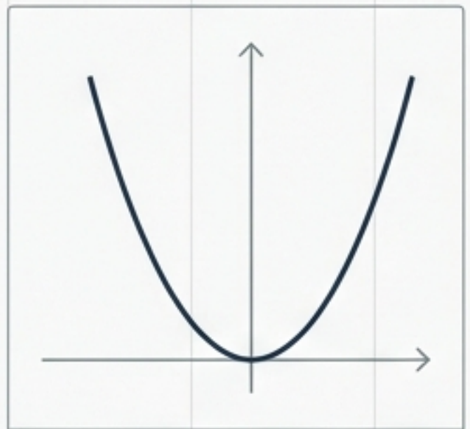
The Hessian (H):

A matrix of second-order partial derivatives.

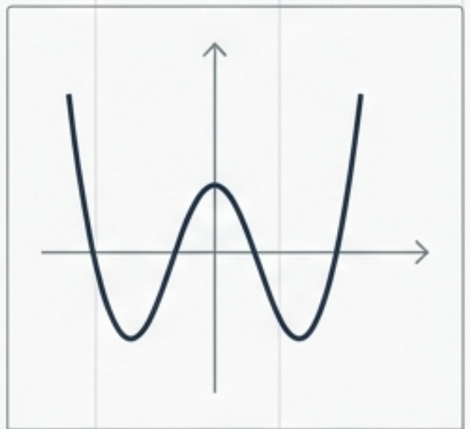
$$H_{ij} = \frac{\partial^2 L}{\partial w_i \partial w_j}$$

Convexity:

- If the function curves upward everywhere (like a bowl), H is Positive Semi-Definite.
- Property: The curve always lies *above* its tangent plane.
- Implication: Optimization is easy; there is only one minimum.



Convex (Positive Curvature)



Non-Convex

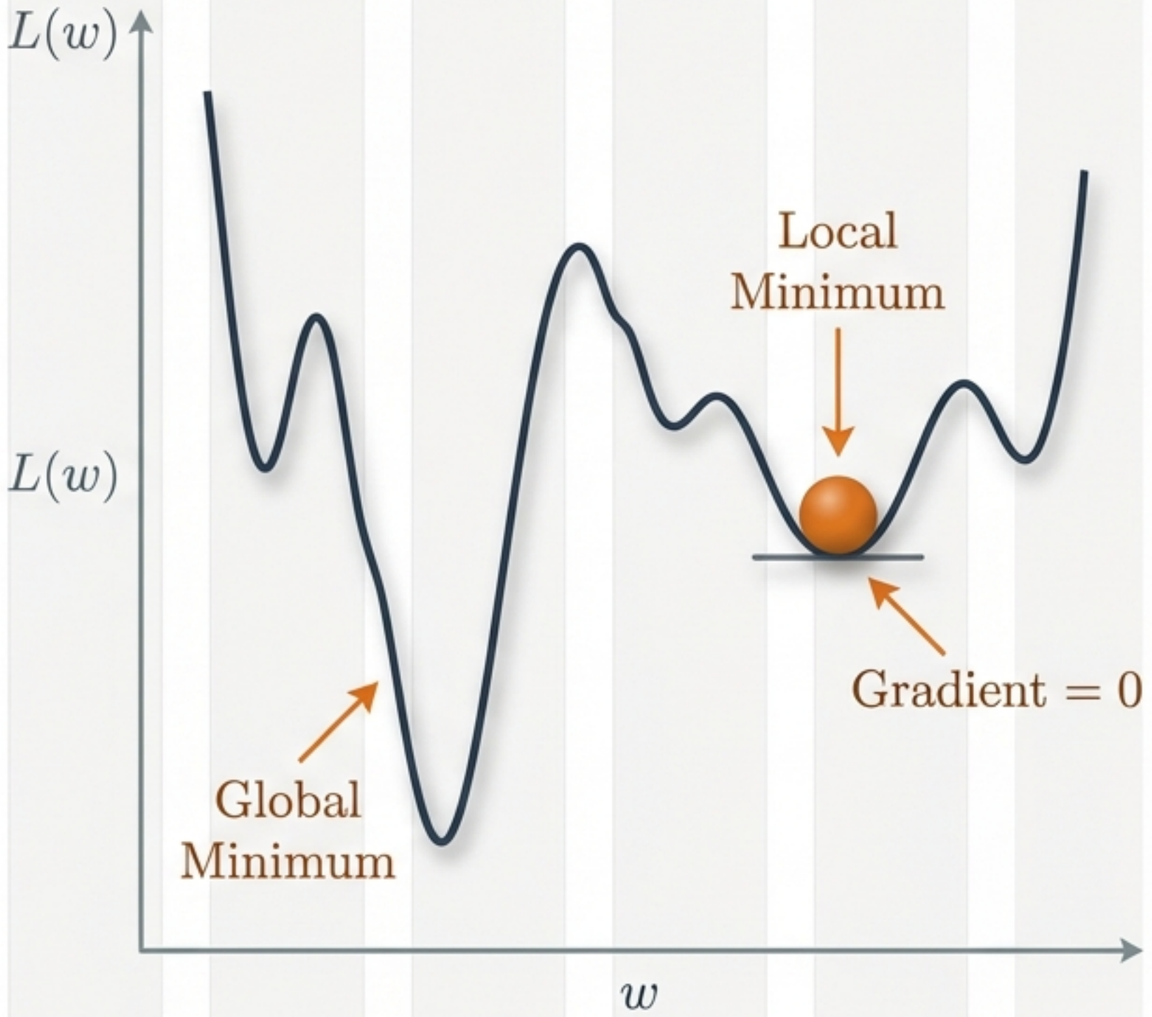
The Trap: Global vs. Local Minima

Convex Functions: Have only one minimum (Global). Descent is guaranteed.

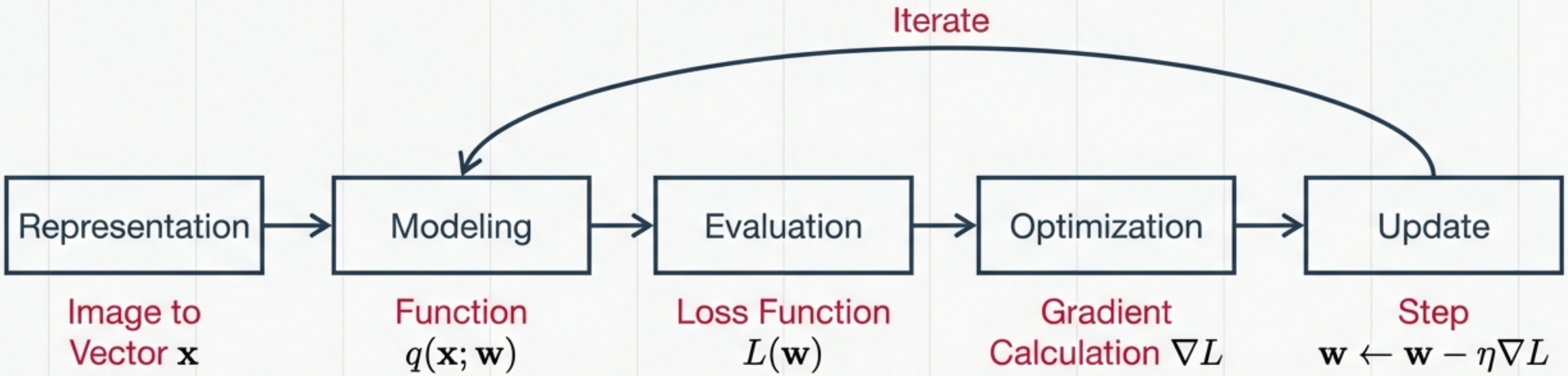
Non-Convex Functions: Have peaks, valleys, and saddle points.

The Trap: At a Local Minimum, the gradient is zero. The model thinks it has finished learning, even though a better solution exists.

Reality: Deep Learning surfaces are highly non-convex. We rely on stochastic noise (SGD) to escape these traps.



Summary: The Learning Pipeline



From Math to Code (PyTorch)

Theory

1. **Forward Pass**
   $y = \text{model}(x)$
   $L = \text{error}(y, \hat{y})$
2. **Backward Pass**
   Compute Gradients ∇L
3. **Update**
   $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L$

Practice

```
# 1. Forward Pass
y_pred = model(x)
loss = loss_fn(y_pred, y_true)

# 2. Backward Pass (Autograd)
loss.backward()

# 3. Update Step
with torch.no_grad():
    w -= learning_rate * w.grad
```

Autograd builds a dynamic computation graph to calculate derivatives automatically, scaling to millions of parameters.

12. PyTorch Code Explanations

12.1 Overview of PyTorch Training

Three main approaches:

1. Manual gradient computation (simple models only)
2. Autograd (automatic differentiation)
3. High-level APIs (nn.Module, optimizers)

12.2 Manual Gradients (Simple Example)

```
import torch

# Generate data: y = 1.5x + 2.73 + noise
x = 10 * torch.randn(100)
y = 1.5 * x + 2.73 + 0.5 * torch.randn(100)

# Initialize parameters
w = torch.randn(1)
b = torch.randn(1)

learning_rate = 0.001
num_steps = 1000

for step in range(num_steps):
    # Forward pass
    y_pred = w * x + b

    # Compute loss
    loss = torch.mean((y_pred - y) ** 2)

    # Manual gradient computation
    w_grad = torch.mean(2 * (y_pred - y) * x)
    b_grad = torch.mean(2 * (y_pred - y))

    # Update parameters
    w = w - learning_rate * w_grad
    b = b - learning_rate * b_grad

print(f"Learned: y = {w.item():.3f}x + {b.item():.3f}")
# Output: y = 1.501x + 2.741
```

Key points:

- We derive gradient formulas manually using calculus
- Only feasible for simple models
- Good for understanding, not for complex architectures

12.3 Autograd: Automatic Differentiation

How it works:

1. PyTorch builds a **computational graph** during forward pass
2. Records all operations on tensors with `requires_grad=True`
3. Applies chain rule automatically during `backward()`
4. Stores gradients in `.grad` attribute

Example computational graph for $L = (wx + b - y)^2$:

```
      L
      ↑
     ()²
      ↑
      (-)
     ↙  ↘
   (+)    y
  ↙  ↘
(×)   b
↙ ↘
w   x
```

12.4 Autograd Implementation

```
import torch

# Generate data
x = 10 * torch.randn(100)
y = 1.5 * x + 2.73 + 0.5 * torch.randn(100)
```

```
# Initialize with gradient tracking
w = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

learning_rate = 0.001
num_steps = 1000

for step in range(num_steps):
    # Forward pass
    y_pred = w * x + b

    # Compute loss
    loss = torch.mean((y_pred - y) ** 2)

    # Backward pass (automatic!)
    loss.backward()

    # Update parameters (no grad tracking)
    with torch.no_grad():
        w -= learning_rate * w.grad
        b -= learning_rate * b.grad

        # IMPORTANT: Zero gradients
        w.grad.zero_()
        b.grad.zero_()

print(f"Learned: y = {w.item():.3f}x + {b.item():.3f}")
```

Key differences from manual:

- Set `requires_grad=True` on parameters
- Call `loss.backward()` instead of computing gradients
- Access gradients via `.grad` attribute
- Must zero gradients after each update

12.5 Why Zero Gradients?

Problem: PyTorch **accumulates** gradients by default.

```
w = torch.tensor([2.0], requires_grad=True)
```

```
loss1 = w ** 2
loss1.backward()
print(w.grad)  # tensor([4.])

loss2 = w ** 2
loss2.backward()
print(w.grad)  # tensor([8.]) ← Accumulated!

w.grad.zero_()
print(w.grad)  # tensor([0.]) ← Reset
```

Solution: Call `zero_()` after each update.

12.6 Nonlinear Functions in PyTorch

Problem: Linear model on nonlinear data fails.

Example: Data follows $y = x^2 - x + 2$

```
# Generate nonlinear data
x = 10 * torch.rand(100) - 5  # Range [-5, 5]
y = x**2 - x + 2 + torch.randn(100)

# Try linear model (will fail)
w = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

for step in range(1000):
    y_pred = w * x + b
    loss = torch.mean((y_pred - y) ** 2)
    loss.backward()

    with torch.no_grad():
        w -= 0.001 * w.grad
        b -= 0.001 * b.grad
        w.grad.zero_()
        b.grad.zero_()

# Linear model achieves high loss
print(f"Linear model loss: {loss.item():.2f}")
```

```
# Output: ~15-20

# Try quadratic model (will succeed)
w0 = torch.randn(1, requires_grad=True)
w1 = torch.randn(1, requires_grad=True)
w2 = torch.randn(1, requires_grad=True)

for step in range(1000):
    y_pred = w0 + w1 * x + w2 * (x ** 2)
    loss = torch.mean((y_pred - y) ** 2)
    loss.backward()

    with torch.no_grad():
        w0 -= 0.001 * w0.grad
        w1 -= 0.001 * w1.grad
        w2 -= 0.001 * w2.grad
        w0.grad.zero_()
        w1.grad.zero_()
        w2.grad.zero_()

# Quadratic model achieves low loss
print(f"Quadratic model loss: {loss.item():.2f}")
# Output: ~1-2

print(f"Learned: y = {w2.item():.2f}x² + {w1.item():.2f}x +
{w0.item():.2f}")
# Output close to: y = 1.00x² - 1.00x + 2.00
```

Lesson: Model architecture must match data complexity.

12.7 High-Level PyTorch: nn.Module

Modern approach: Use PyTorch's high-level APIs.

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define model
class LinearModel(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        self.linear = nn.Linear(1, 1)  # 1 input, 1 output

    def forward(self, x):
        return self.linear(x)

# Create model
model = LinearModel()

# Define loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(1000):
    # Forward pass
    y_pred = model(x)
    loss = criterion(y_pred, y)

    # Backward pass
    optimizer.zero_grad()  # Zero gradients
    loss.backward()        # Compute gradients
    optimizer.step()       # Update parameters

print(f"Learned parameters: {list(model.parameters())}")
```

Advantages:

- Cleaner code
- Built-in optimizers (SGD, Adam, etc.)
- Easy to extend to complex architectures
- Standard in practice

---

13. Convex and Non-Convex Functions

13.1 Definitions

Convex Function

Informal: "Bowl-shaped" - curves upward everywhere.

Formal (1D): Function $f(x)$ is convex if:

$$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha)f(x_2)$$

for all x_1, x_2 and $\alpha \in [0, 1]$.

Geometric interpretation: Line segment connecting any two points on the curve lies **above** the curve.

Examples:

- $f(x) = x^2$ ✓
- $f(x) = e^x$ ✓
- $f(x) = |x|$ ✓
- $f(x) = -\log(x)$ for $x > 0$ ✓

Non-Convex Function

Definition: A function that is not convex.

Characteristics:

- May have multiple local minima
- Line segment between some pairs of points passes **below** curve
- Optimization is more challenging

Examples:

- $f(x) = x^3$ (has inflection point)
- $f(x) = \sin(x)$ (oscillates)
- $f(x) = x^4 - 5x^2 + 4x$ (multiple minima)

13.2 Testing for Convexity

One-Dimensional Test

Condition: $f(x)$ is convex if and only if:

$$\frac{d^2f}{dx^2} \geq 0 \quad \text{for all } x$$

Example 1: $f(x) = x^2$

$$\frac{df}{dx} = 2x, \quad \frac{d^2f}{dx^2} = 2 > 0 \quad \checkmark \text{ Convex}$$

Example 2: $f(x) = x^3$

$$\frac{df}{dx} = 3x^2, \quad \frac{d^2f}{dx^2} = 6x$$

- For $x > 0$: $6x > 0$ (convex)
- For $x < 0$: $6x < 0$ (concave)
- **Not convex everywhere** ×

Multidimensional Test

Condition: $f(\vec{w})$ is convex if and only if the Hessian matrix $H[f(\vec{w})]$ is **positive semi-definite** everywhere.

Positive semi-definite means: All eigenvalues $\lambda_i \geq 0$

Example: $f(w_0, w_1) = 2w_0^2 + 3w_1^2$

Hessian:

$$H = \begin{bmatrix} 4 & 0 \\ 0 & 6 \end{bmatrix}$$

Eigenvalues: $\lambda_1 = 4, \lambda_2 = 6$ (both > 0)

Conclusion: Function is convex ✓

13.3 Convexity and Taylor Series

Connection: The second-order term in Taylor series:

$$\frac{1}{2}\Delta\vec{w}^T H[f(\vec{w})]\Delta\vec{w}$$

For convex functions:

- H is positive semi-definite
- This term is always ≥ 0
- Quadratic approximation lies above the tangent plane

Visualization:

- Convex function: curves upward from tangent
- Non-convex: may curve upward or downward

13.4 Implications for Machine Learning

Convex Loss Functions

Advantages:

- **Single global minimum** (no local minima)
- Gradient descent **guaranteed to converge**
- Final solution independent of initialization
- Reliable optimization

Examples:

- Linear regression with MSE loss
- Logistic regression with log loss
- Support Vector Machines

Non-Convex Loss Functions

Challenges:

- **Multiple local minima**
- Gradient descent may get stuck

- Final solution depends on initialization
- No convergence guarantees

Examples:

- Neural networks (multilayer)
- Deep learning models

Practical strategies:

1. **Multiple random initializations**: Run training several times
2. **Advanced optimizers**: Adam, RMSprop (adaptive learning rates)
3. **Accept local minima**: Often "good enough" for complex problems
4. **Careful initialization**: Xavier, He initialization

13.5 Worked Example: Analyzing Convexity

Function: $f(w_0, w_1) = w_0^2 + 2w_0w_1 + 3w_1^2$

Step 1: Compute partial derivatives

$$\frac{\partial f}{\partial w_0} = 2w_0 + 2w_1$$

$$\frac{\partial f}{\partial w_1} = 2w_0 + 6w_1$$

Step 2: Compute second partial derivatives

$$\frac{\partial^2 f}{\partial w_0^2} = 2$$

$$\frac{\partial^2 f}{\partial w_1^2} = 6$$

$$\frac{\partial^2 f}{\partial w_0 \partial w_1} = 2$$

Step 3: Form Hessian

$$H = \begin{bmatrix} 2 & 2 \\ 2 & 6 \end{bmatrix}$$

Step 4: Compute eigenvalues

Characteristic equation: $\det(H - \lambda I) = 0$

$$(2 - \lambda)(6 - \lambda) - 4 = 0$$

$$\lambda^2 - 8\lambda + 8 = 0$$

$$\lambda = \frac{8 \pm \sqrt{64 - 32}}{2} = \frac{8 \pm \sqrt{32}}{2} = 4 \pm 2\sqrt{2}$$

$$\lambda_1 \approx 6.83, \quad \lambda_2 \approx 1.17$$

Both positive ✓

Conclusion: Function is convex.

Summary of Key Concepts

Core Machine Learning Pipeline

1. **Data representation**: Convert inputs (images) to vectors via rasterization
2. **Feature space**: Each input becomes a point in high-dimensional space
3. **Classification**: Find decision boundary separating point clusters
4. **Parametric model**: Choose function family $\phi(\vec{x}; \vec{w}, b)$
5. **Loss function**: Quantify prediction errors $L(\vec{w}, b)$
6. **Gradient descent**: Iteratively minimize loss via $\vec{w} \leftarrow \vec{w} - \eta \nabla L$
7. **Convergence**: Stop when loss is sufficiently low

Mathematical Tools

Concept	1D	Multidimensional
Rate of change	Derivative $\frac{dL}{dw}$	Gradient $\nabla L(\vec{w})$

Curvature	Second derivative $\frac{d^2L}{dw^2}$	Hessian $H[L(\vec{w})]$
Convexity test	$\frac{d^2L}{dw^2} \geq 0$	H positive semi-definite
Update rule	$w \leftarrow w - \eta\frac{dL}{dw}$	$\vec{w} \leftarrow \vec{w} - \eta\nabla L$

PyTorch Implementation Levels

Approach	Complexity	Use Case
Manual gradients	Low	Learning, simple models
Autograd	Medium	Custom architectures
nn.Module + optimizers	High	Production, complex models

Optimization Landscape

Convex functions:

- ✓ Single global minimum
- ✓ Guaranteed convergence
- ✓ Initialization-independent
- Example: Linear regression

Non-convex functions:

- ✗ Multiple local minima
- ✗ May get stuck
- ✗ Initialization-dependent
- Example: Neural networks

Study Guide & Practice Problems

Conceptual Questions

1. Explain why rasterization allows us to view images as points in feature space.
2. What is the geometric interpretation of a classifier?
3. Why do we follow the negative gradient during training?
4. What does it mean when the gradient is zero?

5. Explain the difference between partial derivatives and the full gradient.
6. Why is the gradient perpendicular to level contours?
7. What information does the Hessian matrix provide?
8. Why might a non-convex loss function cause problems in training?

Computational Exercises

1. **Compute gradient**: For $L(w_0, w_1) = w_0^2 + w_0 w_1 + 2w_1^2$, find ∇L at $(1, 2)$.
2. **Gradient descent step**: Starting at $\vec{w} = [2, 3]^T$ with $\eta = 0.1$, perform one update step for the loss function above.
3. **Hessian matrix**: Compute the Hessian of $L(w_0, w_1, w_2) = w_0^2 + w_1^2 + w_2^2 + w_0 w_1$.
4. **Convexity test**: Determine if $f(w) = w^4$ is convex for all w.
5. **Taylor approximation**: For $f(x) = \sin(x)$, write the second-order Taylor approximation around $x = 0$.

Programming Exercises

1. Implement gradient descent from scratch (no PyTorch) for $f(x) = (x - 3)^2$.
2. Use PyTorch Autograd to train a quadratic model on synthetic data.
3. Visualize the loss surface for $L(w_0, w_1) = w_0^2 + 3w_1^2$ as a 3D plot.
4. Implement and compare linear vs. polynomial models on nonlinear data.
5. Create an animation showing gradient descent optimization path on a 2D loss surface.

References & Further Reading

Textbook Sections

- Section 3.1: Geometrical view of image classification
- Section 3.2: Error/loss function
- Section 3.3: Minimizing loss functions
- Section 3.4: Local approximation for loss function
- Section 3.5: PyTorch code
- Section 3.6-3.7: Convex and non-convex functions

Key Equations to Remember

Gradient descent update:

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \eta \nabla_{\vec{w}} L(\vec{w}^{(t)})$$

Gradient (multidimensional):

$$\nabla L(\vec{w}) = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

Taylor series (second-order):

$$L(\vec{w} + \Delta\vec{w}) \approx L(\vec{w}) + \nabla L^T \Delta\vec{w} + \frac{1}{2}\Delta\vec{w}^T H \Delta\vec{w}$$

Convexity condition:

$$H[L(\vec{w})] \succeq 0 \text{ (all eigenvalues} \geq 0)$$

Appendix: Common Pitfalls & Debugging

Training Issues

Problem: Loss not decreasing

- **Check**: Learning rate too high → reduce η
- **Check**: Wrong gradient computation → verify manually
- **Check**: Forgot to zero gradients → add `grad.zero_()`

Problem: Loss decreasing very slowly

- **Check**: Learning rate too low → increase η
- **Check**: Poor initialization → try different random seed
- **Check**: Model too simple → increase capacity

Problem: Loss exploding (NaN values)

- **Check**: Learning rate too high → reduce significantly
- **Check**: Gradient explosion → clip gradients
- **Check**: Numerical instability → check for divide by zero

PyTorch-Specific Issues

Problem: "Trying to backward through the graph a second time"

- **Solution**: Set `retain_graph=True` or restructure computation

Problem: Gradients accumulating incorrectly

- **Solution**: Call `optimizer.zero_grad()` or `param.grad.zero_()` before each backward pass

Problem: Tensors on different devices

- **Solution**: Move all tensors to same device with `.to(device)`

End of Chapter 3 Lecture Notes

These notes synthesize key concepts from the textbook with practical implementation guidance and worked examples for graduate-level machine learning education.